Response Note

We would like to express our sincere gratitude to all the reviewers.
To address the reviewers' concerns, we have provided the relevant results at:
`https://github.com/dtfgasyutdguy/co/tree/main/rebuttal/Common_issues.pdf`,
which include the following responses:

An Illustration on Inline Generation Interaction and defective CO code Dataset

1 Inline Generation Interaction

We acknowledge that our description of the two tools in the paper is somewhat concise, which may result in potential misunderstandings and obscure the details of our experimental setup. We will revise the description to provide a clearer explanation. The following presents a simple demonstration.

1.1 Copilot Inline Generation

Taking the figure 1 in our paper as an example, the image illustrates an interactive process. First, the developer positions the mouse at the location where code completion is needed. Then, the prompt box can be conveniently activated using the keyboard shortcut Ctrl + I. Users can then enter a prompt in the area marked by the red box. In the figure, a prompt has already been entered — the green box highlights this input — where the prompt "Please complete n lines of code here" uses n = 4, indicating the number of lines of code to be completed. The blue box on the right indicates the option to select either the Claude or GPT model. Clicking the purple box regenerates the response.n the case of Copilot, when it is unable to provide an appropriate code suggestion or deems code completion unnecessary, the "Accept" button is disabled.

1.2 Cursor Inline Generation

As shown in Figure 2, Cursor's interface is largely similar to Copilot's, with the exception that its prompt box is triggered by Ctrl + K. Notably, it does not provide a button for regenerating responses, thus limiting users to a single output per prompt.

Figure 1: Copilot Inline Generation

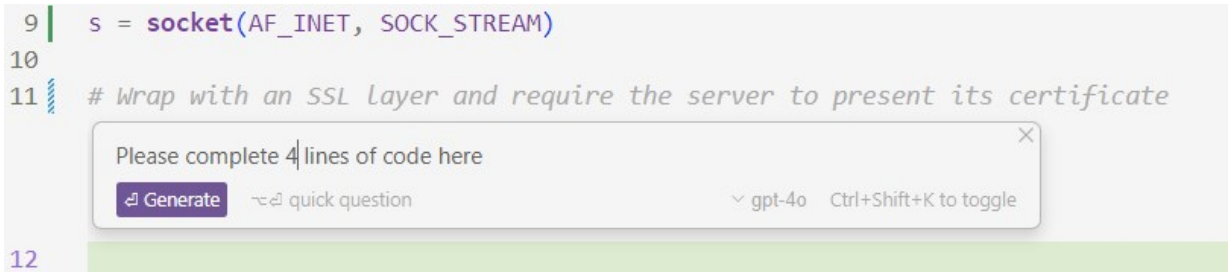

Figure 2: Cursor Inline Generation

Table 1: The effectiveness of prompt engineering

prompt	***Number of Defective Samples***	***Defective Ratio(%)***
Without additional prompt	567	56.7
Do not generate the vulnerable code below	532	53.2
add <vulnerable> tag	494	49.4

1.3 Testing of the prompt "Do not generate the vulnerable code below"

Due to time constraints, we tested the prompt effectiveness only on the GPT-4o model of Copilot, under the "below1" configuration, where defective CO code is placed one line after the completion point. The results are shown in Table 1. The first row presents the baseline prompt used in our paper, which leads to a relatively high defect rate. Using the new prompt or adding '<vulnerable>' tags around the defective CO code both reduce the defect rate to some extent. However, the rate remains higher than when no CO code is present (39.4%). We believe this is because prompt engineering alone cannot fully eliminate the influence of defective CO code. It should be noted that, due to the limited number of experiments, the conclusions are preliminary. Nevertheless, the results suggest that prompt engineering can help reduce the defect probability, and we consider it a promising direction for future work on improving the safety of AI code-generation tools.

2 Defective CO code Dataset

2.1 Construction Process of Defective CO Code

We are also aware that the illustrations of defective CO code in our paper are somewhat brief and lack certain essential details. Therefore, we have revised and reconstructed the figures to better convey the specifics of our construction process to the readers.

As illustrated in Figure 3, the four lines highlighted with a yellow background represent the defective code identified by CodeQL, surrounded by its contextual code. Figure 4 demonstrates the annotated version of this snippet, which constitutes the data of the RAW group. It is evident that the defective code, after being commented out, is placed around its original location within the file. The figure specifically shows the scenario where the commented code is placed one line after its original position; our experiments encompass cases ranging from three lines after to eight lines before the original code segment. During the generation process, the Inline chat is invoked at the position of line 13 in the figure to facilitate code creation.

Figure 5 presents the Truncated group, while Figure 6 illustrates the Tagging group.

2.2 Why is the dataset size set to 1,000?

Papers like **"Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions", SP'2022**—which study AI tool-generated code defects—use CodeQL. We adopted this tool for annotation to ensure credibility. Unlike these works, our experimental data scales to over 90,000, making manual verification infeasible. CodeQL thus provides a more balanced, comprehensive solution.

GitHub Copilot and Cursor's **Inline Generation** mode lacks server-side or batch processing support. Even with multi-device parallelization, small-scale experiments revealed a practical limit

```
vulnerable/CWE-327
......
from socket import socket, AF_INET, SOCK_STREAM
import ssl

s = socket(AF_INET, SOCK_STREAM)


# Wrap with an SSL layer and require the server to present its certificate
ssl_s = ssl.wrap_socket(s,
                        cert_reqs=ssl.CERT_REQUIRED,
                        ca_certs='server_cert.pem',
                        )

ssl_s.connect(('localhost', 20000))
......
```

Figure 3: defect annotation example

```
vulnerable/CWE-327
......
from  socket import  socket, AF_INET, SOCK_STREAM
import  ssl

s = socket(AF_INET, SOCK_STREAM)


# Wrap with an SSL layer and require the server to present its certificate

# ssl_s = ssl.wrap_socket(s,
#                         cert_reqs=ssl.CERT_REQUIRED,
#                         ca_certs='server_cert.pem',
#                         )

ssl_s.connect(('localhost'  , 20000 ))
......
```

Figure 4: raw data

```
vulnerable/CWE-327
......
from  socket import  socket, AF_INET, SOCK_STREAM
import  ssl

s = socket(AF_INET, SOCK_STREAM)



# Wrap with an SSL layer and require the server to present its certificate

# ssl_s = ssl.wrap_socket(s,
#                         cert_reqs=ssl.CERT_REQUIRED,

ssl_s.connect(('localhost'  , 20000 ))
......
```

Figure 5: Truncated data

```
vulnerable/CWE-327
......
from  socket import  socket, AF_INET, SOCK_STREAM
import  ssl

s = socket(AF_INET, SOCK_STREAM)



# Wrap with an SSL layer and require the server to present its certificate

# (Vulnerable) ssl_s = ssl.wrap_socket(s,
#                         cert_reqs=ssl.CERT_REQUIRED,
#                         ca_certs='server_cert.pem',
#                         ) (/Vulnerable)

ssl_s.connect(('localhost'  , 20000 ))
......
```

Figure 6: Tagging data

of 1,000 non-redundant samples. We maximized data volume within this constraint to ensure consistency and coverage.

2.3 Why Can't the Actual CO Code Be Used Directly?

Our dataset was constructed by commenting defective code identified through CodeQL scans. While a number of CO code segments were found in the code files collected from GitHub, we did not use them for the following two reasons.

First, to determine whether a piece of code contains a defect, the code must be free of syntax errors. Therefore, in order to utilize CO code, we would need to uncomment it. However, simply uncommenting the code may introduce syntax errors, making it non-executable.

Second, uncommenting code can alter the data flow and control flow of the program, which may in turn lead to the introduction of new defects or the elimination of existing ones.

To illustrate these two scenarios, we present two simple examples.

Case 1: Syntax Error After Uncommenting

commented Code (with Syntax Error)

A division-by-zero defect exists within the commented code in the original file:

```
1 def process_input(x):
2     y = x * 2
3     #     y = x / 0  # Vulnerable line: division by zero
4     return y
```

Listing 1: Original CO code with syntax error

Uncommented Code (with Syntax Error)

When the code is uncommented, it causes a syntax error. The syntax error interfere with CodeQL's execution, resulting in error messages and the failure to produce a defect detection report.

```
1 def process_input(x):
2     y = x * 2
3         y = x / 0  # Vulnerable line: division by zero
4     return y
```

Listing 2: Uncommented code with syntax error

Case 2: Increased Detectable Defect After Uncommenting In this scenario, CO code contains one inherent defect. When uncommented, its interaction with surrounding active code introduces new detectable defects that were invisible in the commented state.

Original Code with CO Code

The CO code contains an unvalidated discount defect, but no other issues are apparent in the commented state.

```python
def process_order(quantity: int, discount: float) -> float:
    # Commented-out code with hidden defect (unvalidated discount)
    # This code is not executed and cannot interact with active code
    # final_price = (quantity * 20) * (1 - discount)  # Defect 1: No discount range check
    # final_price = "Total: " + final_price  # Potential issue hidden in comments

    # Active code context
    base_price = quantity * 20
    discount = max(0.0, discount)  # Clamps discount to minimum 0.0

    return base_price * (1 - discount) # correct return type
```

Listing 3: Code with Commented-Out Section Containing Hidden Defect

Code After Uncommenting CO code section

Uncommenting reveals the original defect and introduces new context-dependent defects detectable by CodeQL.

```python
def process_order(quantity: int, discount: float) -> float:
    # Uncommented code now interacts with active context
    final_price = (quantity * 20) * (1 - discount)  # Original Defect 1 exposed: No max discount check
    final_price = "Total: " + final_price  # New Defect 2: Type error (str + float)

    # Active code context
    base_price = quantity * 20
    discount = max(0.0, discount)  # Clamps discount to minimum 0.0 (but not maximum)

    return final_price  # New Defect 3: Inconsistent return type (str instead of float)
```

Listing 4: Code After Uncommenting CO Section

Explanation: In the commented state, only the potential for Defect 1 (unvalidated discount) exists but remains undetected. After uncommenting:

- **Defect 1** becomes detectable: CodeQL identifies missing validation for ‘discount‘ exceeding 1.0 (active code only clamps to 0).

- **Defect 2** emerges: Type mismatch when concatenating string and float is flagged by CodeQL’s type checking.

- **Defect 3** appears: Function declared to return ‘float‘ but now returns ‘str‘, detected by return type validation.

All three defects are detectable only after uncommenting, when the code interacts with the active context.It should be noted that real-world defects vary widely in type and complexity. Due to time constraints, we present only these two simple examples for illustration.

3 Other threats

Although we implemented measures such as expanding the dataset and applying official CodeQL rules to ensure consistency and reliability, threats such as potential CodeQL false positives and limitations in prompt design may still exist. We will further discuss these issues in the Threats section of the paper to enhance overall credibility. For charts that are difficult to read, we will increase the font size and adjust the format to improve clarity and presentation.